17장 생성자 함수에 의한 객체 생성

- 10장 객체 리터럴({...}) 외에 다양한 방법으로 객체를 생성할 수 있으나, 생성자 함수를 사용하여 객체를 생성할 수도 있다.

17.1 Object 생성자 함수

- new 연산자와 함께 Object 생성자 함수를 호출하여 빈 객체를 생성 및 반환한다.
  - 빈 객체를 생성한 후 프로퍼티 또는 메서드를 추가하여 객체를 완성할 수 있다.

```
// 빈 객체의 생성
const person = new Object();

// 프로퍼티 추가
person.name = 'Lee';
person.sayHello = function () {
  console.log('Hi! My name is ' + this.name);
};

console.log(person); // {name: "Lee", sayHello: f}
person.sayHello(); // Hi! My name is Lee
```

- 생성자 함수 : new 연산자와 함께 호출하여 객체(인스턴스)를 생성하는 함수
- 인스턴스 : 생성자 함수에 의해 생성된 객체

- 자바스크립트는 Object 생성자 함수 외에도 String, Nunber, Boolean, Function, Array, Date, RegExp, Promise 등의 빌트인 생성자 함수를 제공한다.

17.2 생성자 함수

1. 객체 리터럴에 의한 객체 생성 방식의 문제점

```
const circle1 = {
  radius: 5,
  getDiameter() {
    return 2 * this.radius;
  }
};

console.log(circle1.getDiameter()); // 10

const circle2 = {
  radius: 10,
  getDiameter() {
    return 2 * this.radius;
  }
};

console.log(circle2.getDiameter()); // 20
```

- 객체 리터럴에 의한 객체 생성 방식은 단 하나의 객체만 생성한다.
- 따라서 동일한 프로퍼티를 갖는 객체를 여러 개 생성해야 하는 경우 매번 같은 프로퍼티를 기술해야 하기 때문에 비효율적이다.

2. 생성자 함수에 의한 객체 생성 방식의 장점

- 생성자 함수에 의한 객체 생성 방식은 마치 객체(인스턴스)를 생성하기 위한 템플릿(클래스)처럼 생성자 함수를 사용하여 프로퍼티 구조가 동일한 객체 여러 개를 간편하게 생성할 수 있다.

```
function Circle(radius) {
  // 생성자 함수 내부의 this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// 인스턴스의 생성
```

```
const circle1 = new Circle(5); // 반지름이 5인 Circle 객체를 생성
const circle2 = new Circle(10); // 반지름이 10인 Circle 객체를 생

console.log(circle1.getDiameter()); // 10
console.log(circle2.getDiameter()); // 20
```

this

this는 객체 자신의 프로퍼티나 메서드를 참조하기 위한 자기 참조 변수다.

this가 가리키는 값, 즉 this 바인딩은 함수 호출 방식에 따라 동적으로 결정된다.

```
// 함수는 다양한 방식으로 호출될 수 있다.
function foo() {
  console.log(this);
}

// 일반적인 함수로서 호출
// 전역 객체는 브라우저 환경에서는 window, Node.js 환경에서는 global을 가리킨다.
foo(); // window

const obj = { foo }; // ES6 프로퍼티 축약 표현

// 메서드로서 호출
obj.foo(); // obj

// 생성자 함수로서 호출
const inst = new foo(); // inst
```

- new 연산자와 함께 호출하면 해당 함수는 생성자 함수로 동작한다.
    - 만약 new 연산자가 없다면 일반 함수로 동작한다.

3. 생성자 함수의 인스턴스 과정

1. 인스턴스 생성 ⇒ **필수**
2. 생성된 인스튼스를 초기화 ⇒ **옵션**

```
// 생성자 함수
function Circle(radius) {
    // 1. 암묵적으로 인스턴스가 실행되고 this에 바인딩된다.
    console.log(this); // Circle {}

  // 인스턴스 초기화
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// 인스턴스 생성
const circle1 = new Circle(5); // 반지름이 5인 Circle 객체를 생성
```

⇒ 위 생성자 함수에서 인스턴스를 생성하고 반환하는 코드는 **암묵적으로** 진행된다.

1. 인스턴스 생성과 this 바인딩

- 암묵적으로 빈 객체 생성 ⇒ 미완성인 인스턴스 ⇒ this에 바인딩
    - this가 생성자 함수가 생성할 인스턴스를 가리키는 이유

💡 **바인딩**이란?
식별자와 값을 연결하는 과정
this 바인딩은 this(키워드로 분류되지만 식별자 역할을 함)와 this가 가리킬 객체를 바인당하는 것
ex) 변수 선언은 변수 이름(식별자)과 확보된 메모리 공간의 주소를 바인딩하는 것

2. 인스턴스 초기화

- this에 바인딩되어 있는 인스턴스에 프로퍼티나 메서드를 추가하고,

  생성자 함수가 인수로 전달받은 초기값을 인스턴스 프로퍼티에 할당하여 초기화하거나 고정값을 할당

```
function Circle(radius) {
    // 1. 암묵적으로 인스턴스가 실행되고 this에 바인딩된다.

  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}
```

3. 인스턴스 반환

- 생성자 함수 내부에서 모든 처리가 끝나면,

  완성된 인스턴스가 바인딩된 this를 암묵적으로 반환한다.

```
function Circle(radius) {
  // 1. 암묵적으로 빈 객체가 생성되고 this에 바인딩된다.

  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
```

```
  };

  // 3. 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
}

// 인스턴스 생성. Circle 생성자 함수는 암묵적으로 this를 반환한다.
const circle1 = new Circle(5);
console.log(circle); // Circle {radius: 1, getDiameter: f}
```

- 만약 this가 아닌 다른 객체를 명시적으로 반환하면 this가 반환되지 못하고 return 문에 명시한 객체가 반환된다.

```
function Circle(radius) {
  // 1. 암묵적으로 빈 객체가 생성되고 this에 바인딩된다.

  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };

  // 3. 명시적으로 객체를 반환하면 암묵적인 this 반환이 무시된다.
  return {}
}

// 인스턴스 생성. Circle 생성자 함수는 명시적으로 반환한 객체를 반환한다.
const circle1 = new Circle(5);
console.log(circle); // {}
```

- 하지만 명시적으로 원시 값을 반환하면 원시 값 반환은 무시되고 암묵적으로 this가 반환된다.

```
function Circle(radius) {
  // 1. 암묵적으로 빈 객체가 생성되고 this에 바인딩된다.
```

```
  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };

  // 3. 명시적으로 원시 값을 반환되면 암묵적으로 this가 반환된다.
  return 100;
}

// 인스턴스 생성. Circle 생성자 함수는 암묵적으로 this를 반환한다.
const circle1 = new Circle(5);
console.log(circle); // Circle {radius: 1, getDiameter: f}
```

- 이처럼 생성자 함수 내부에서 **명시적으로** this가 아닌 다른 값을 반환하는 것은 생성자 함수의 기본 동작을 훼손한다.
    - 생성자 내부에서 return 문을 **반드시 생략**해야 한다.

4. 내부 메서드 [[Call]]과 [[Contruct]]

- 함수는 객체이기에 일반 객체와 동일하게 동작할 수 있다.
    - 일반 객체가 가지고 있는 내부 슬롯과 내부 메서드를 모두 갖고 있기 때문

```
// 함수는 객체다.
function foo() {}

// 따라서 프로퍼티를 소유할 수 있고,
foo.prop = 10;

// 메서드도 소유할 수 있다.
foo.method = function() {
    console.log(this.prop);
}
```

```
foo.method(); // 10
```

- 그러나 함수는 객체이면서 일반 객체와는 다르다.
  - 일반 객체와 달리 **_호출 가능**하기 때문!
    - 내부 슬롯과 내부 메서드 외에 추가로 함수 객체만을 위한 [[Environment]], [[FormalParameters]] 등의 내부 슬롯과 [[Call]], [[Construct]]같은 내부 메서드를 추가로 가지고 있다.
- 함수가 (1)일반 함수로서 호출되면 함수 객체의 내부 메서드 [[Call]]이 호출되고, (2)new 연산자와 함께 생성자 함수로서 호출되면 내부 메서드 [[Construct]]가 호출된다.

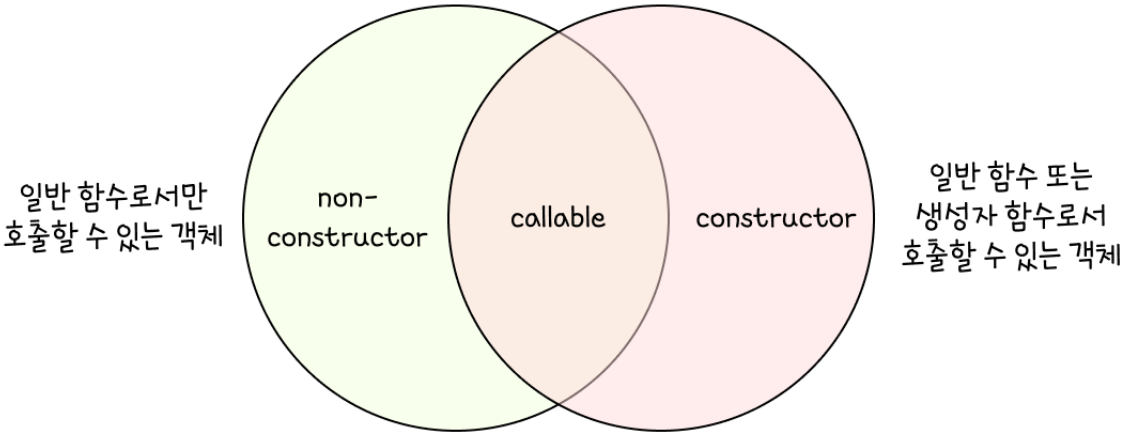
```
function foo() {}

// 일반적인 함수로서 호출: [[Call]]이 호출된다.
foo();

// 생성자 함수로서 호출: [[Construct]]가 호출된다.
new foo();
```

- **callable** : 내부 메서드 [[Call]]을 갖는 함수 객체

  ⇒ 호출할 수 있는 객체(함수)
- **constructor** : 내부 메서드 [[Construct]]를 갖는 함수 객체

  ⇒ 생성자 함수로서 호출할 수 있는 함수
- **non-constructor** : 내부 메서드 [[Construct]]를 갖지 않는 함수 객체

  ⇒ 객체를 생성자 함수로서 호출할 수 없는 함수



https://velog.io/@dnr6054/deepdive-constructor

💡 **함수 객체**는 callable이면서 constructor이거나, callable이면서 non-constructor다
즉, 모든 함수 객체는 호출할 수 있지만 모든 함수 객체를 생성자 함수로서 호출할 수 있는 것은 아니다.

5. constructor와 non-constructor

자바스크립트 엔진은 함수 정의를 평가하여 함수 객체를 생성할 때 함수 정의 방식에 따라 구분

- **constructor** : 함수 선언문, 함수 표현식, 클래스(클래스도 함수다)
- **non-constructor** : 메서드(**ES6 메서드 축약 표현**), 화살표 함수

```
// 일반 함수 정의: 함수 선언문, 함수 표현식
function foo() {}
const bar = function () {};
// 프로퍼티 x의 값으로 할당된 것은 일반 함수로 정의된 함수다. 이는 메서드로
const baz = {
  x: function () {}
};

// 일반 함수로 정의된 함수만이 constructor이다.
```

```
new foo(); // -> foo {}
new bar(); // -> bar {}
new baz.x(); // -> x {}

// 화살표 함수 정의
const arrow = () => {};
new arrow(); // TypeError: arrow is not a constructor

// 메서드 정의: ES6의 메서드 축약 표현만을 메서드로 인정한다.
const obj = {
  x() {}
};

obj.x(); // TypeError: obj.x is not a constructor
```

6. new 연산자

- new 연산자와 함께 함수를 호출하면 해당 함수는 생성자 함수로 동작

  ⇒ [[Call]]이 아닌 [[Construct]]가 호출(단, constructor이어야 한다)
- 반대로 new 연산자 없이 생성자 함수를 호출하면 일반 함수로 호출

생성자 함수는 일반적으로 파스칼 케이스(첫 문자는 대문자)로 명명하여
일반 함수와 구별할 수 있도록 할 것

7. new.target

- new 연산자와 함께 생성자 함수로서 호출되면 함수 내부의 new.target은 함수 자신을 가리킨다.
- new 연산자 없이 일반 함수로서 호출된 함수 내부의 new.target은 undefined다.

```
// 생성자 함수
function Circle(radius) {
```

```
  // 이 함수가 new 연산자와 함께 호출되지 않았다면 new.target은 undefi
  if(!new.target) {
    // new 연산자와 함께 생성자 함수를 재귀 호출하여 생성된 인스턴스를 반
    return new Circle(radius);
  }

  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// new 연산자 없이 생성자 함수를 호출하여도 new.target을 통해 생성자 함
const circle = Circle(5);
console.log(circle.getDiameter());
```

- 대부분의 빌트인 생성자 함수는 new 연산자와 함께 호출되었는지를 확인 한 후 적절한 값을 반환

💡 **스코프 세이프 생성자 패턴**
- new.target은 ES6에서 도입된 최신 문법으로 IE에서는 지원하지 않는다.
- new.target을 사용할 수 없는 상황이라면 스코프 세이프 생성자 패턴 사용 가능

```
// Scope-Safe Constructor Pattern
function Circle(radius) {
  // 이 함수가 new 연산자와 함께 호출되지 않았다면 this는 전역 객체 win
  if(!(this instance of Circle)) {
    // new 연산자와 함께 생성자 함수를 재귀 호출하여 생성된 인스턴스를 반
    return new Circle(radius);
  }

  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
```

```
  };
}

// new 연산자 없이 생성자 함수를 호출하여도 new.target을 통해 생성자 함
const circle = Circle(5);
console.log(circle.getDiameter());
```